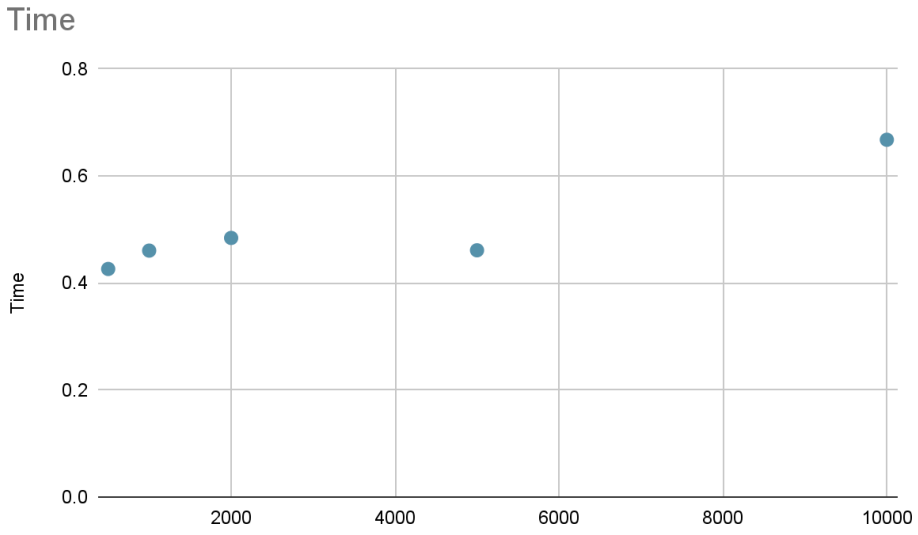
Algorithm:
We used a brute force `O(mn)` algorithm to find the string match. So for each index of sample, we check if starting from that index, we can match the signature.

Parallelisation strategy:
For each (sample, signature) pair, we allocate a block with 128 threads to that pair. So the computation for each pair runs in parallel on the gpu. The work is divided equally among the 128 threads, i.e. each thread checks `samples.size() / 128` indices. Each block has a shared array, that we use to store, for each thread, the smallest matching index found. Then we use the `reducedShared` strategy in Lab 3 to find the smallest index. Then we just have to find the sum of the `phread33` score. We used the same `reducedShared` strategy to find the sum of the `phread33` score, then divide by the length of the signature.

Grid and block dimension:
`gridDim = {samples.size(), signatures.size(), 1}`
`blockDim = {128, 1, 1}`
This way, each (sample, signature) pair gets 1 block to run the string matching algorithm. We allocate 128 threads only, so that the overhead from __syncthreads is lesser.

Memory:
All memory needed by the kernel is transferred to the GPU before the kernel is launched. After the kernel is done, the memory storing the match confidence score is transferred back. Shared memory is used to store the smallest matching index found by each thread, and the intermediate data needed to do the reducedShared algorithm in Lab 3. So we have a `__shared__ int arr[128]` for each block

Changing sequence length
Test command: `./gen_sig 1000 N N 0.001 > t1.fasta`
`./gen_sample t1.fasta 900 100 1 5 10*N 10*N 0 60 0. 001 > t1.fastq`
N: length of signature, Node: h100-96

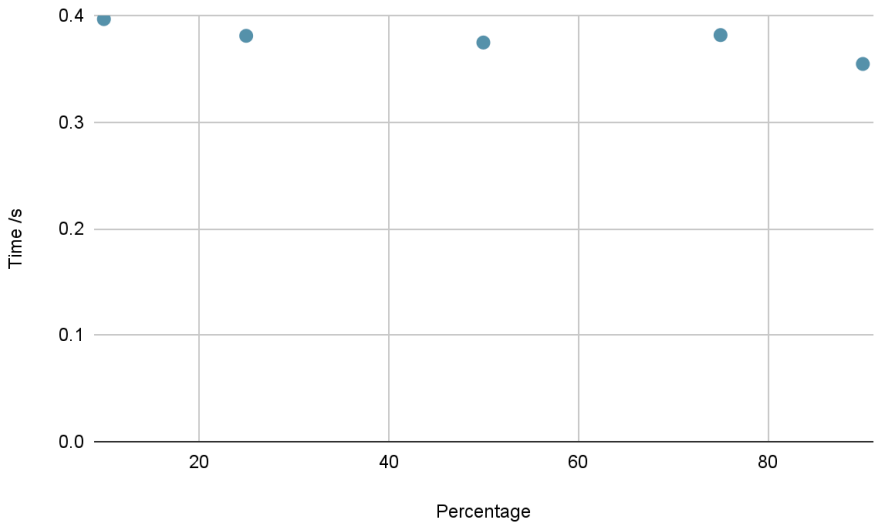


As the signature and sample length increased, the execution time increased, as there is more character matching to be done.

Changing percentage of signatures matching samples

Test command: `./gen_sig 1000 1000 1000 0.001 > t1.fasta`
`./gen_sample t1.fasta X Y 1 5 10000 10000 0 60 0.001 > t1.fastq`

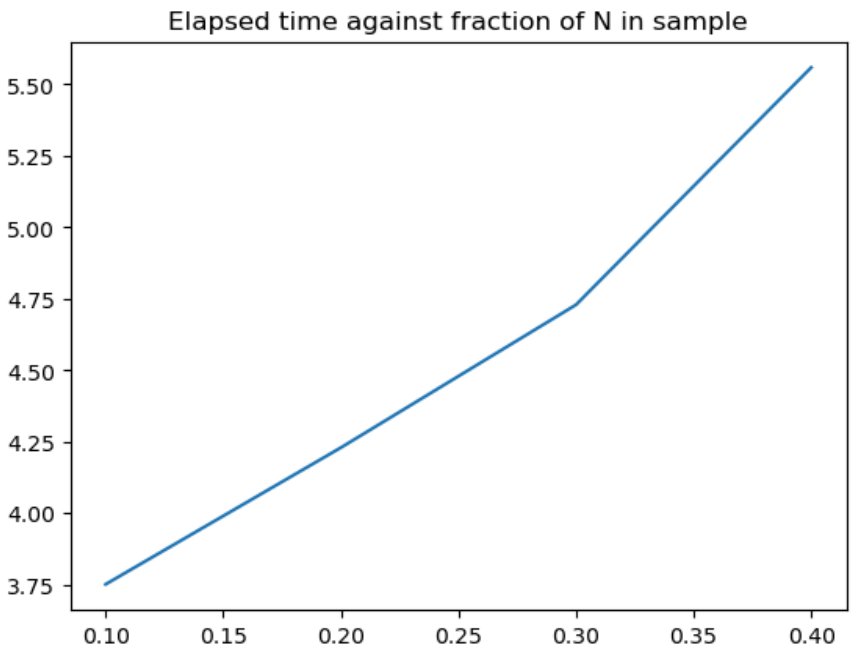
X + Y = 1000, and they are varied to get the desired percentage of samples with virus signatures



There seems to be no significant difference when the percentage of samples with viruses is increased. This could be due to our algorithm not terminating when detecting a match. our algorithm, a block of 128 threads, checks indexes [0..127], [128..255] and so on. Even if a match is detected in range [0..127], the kernel goes on and checks other ranges. Hence, the time taken to process a sample with viruses and sample with no viruses is the same.

Changing fraction of N characters in sample

Test command: `./gen_sig 1000 3000 3100 0.1 > t1.fasta./gen_sample t1.fasta 2000 20 1 1 10000 10000 10 10 X > t1.fastq`

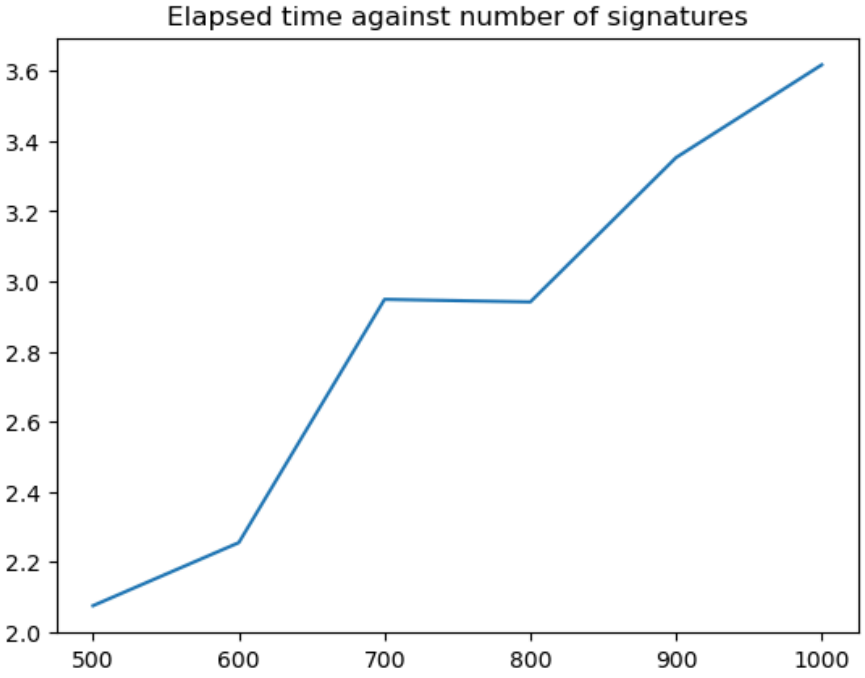


As the fraction of N in the sample increases, on average, at any given index, our algorithm will test a larger amount of sample characters before encountering a mismatch and breaking the loop. The expected characters until encountering a non-N is 1/1-p, explaining the shape of the increase.

Changing number of signatures

Test command: `./gen_sig X 3000 3100 0.1 > t1.fasta`

`./gen_sample t1.fasta 2000 20 1 1 10000 10000 10 10 0.1 > t1.fastq`



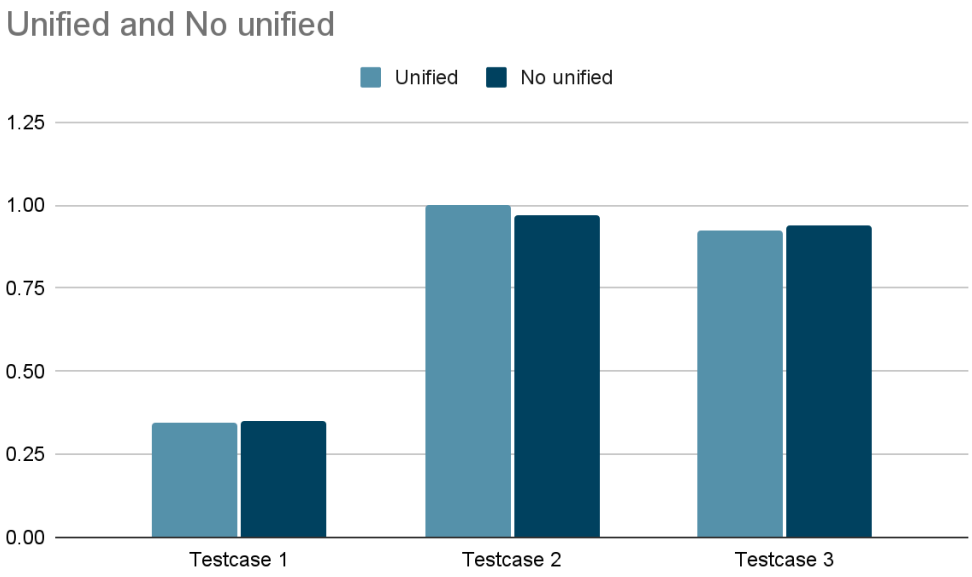
As the number of signatures increases, so does elapsed time. This increase is roughly linear because each signature is processed individually. As the number of signatures increases, there are more (sample, signature) pairs to process.

Optimisation 1:
At first, we used unified memory for some of the data we needed to store. We hypothesize that using unified memory is slower, as the GPU would use the RAM on the motherboard, instead of GPU VRAM which is closer and accessed faster. From our results, the difference is negligible. The reason could be that accessing unified memory is slower but is allocated faster, but when using device memory, memory has to be explicitly transferred to the gpu which takes time, hence the differences balances out.

Node: h100-96
Testcase 1: `./gen_sig 1000 1000 1000 0.001 > t1.fasta`
`./gen_sample t1.fasta 900 100 1 5 10000 10000 0 60 0.0001 > t1.fastq`

Test case 2: `./gen_sig 1000 1000 1000 0.001 > t1.fasta`
`./gen_sample t1.fasta 900 100 1 5 100000 100000 0 60 0.0001 > t1.fastq`

Test case 3: `./gen_sig 1000 10000 10000 0.001 > t1.fasta`
`./gen_sample t1.fasta 900 100 1 5 100000 100000 0 60 0.0001 > t1.fastq`

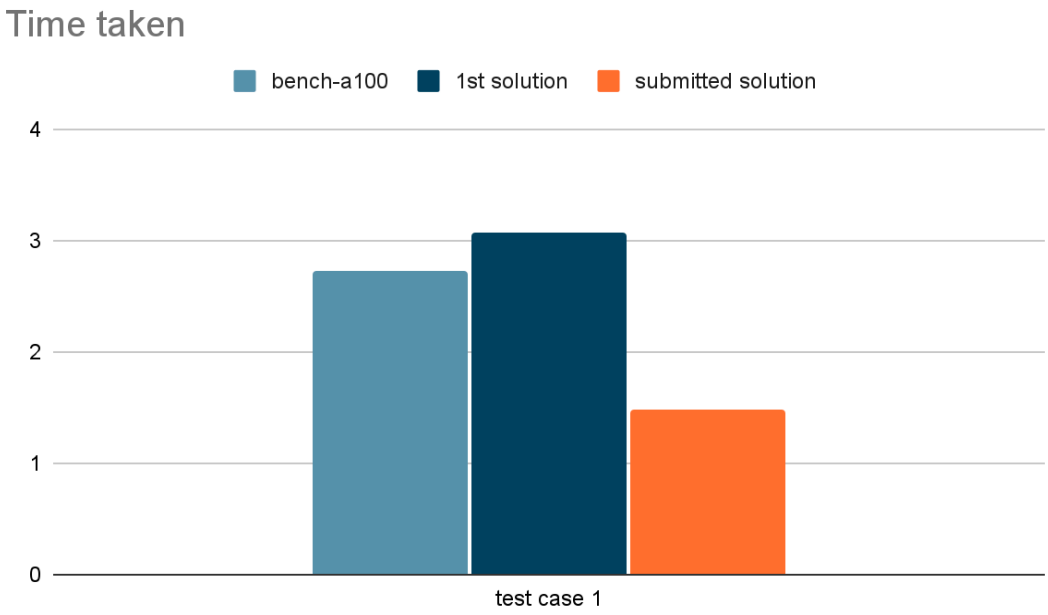


Optimisation 2:
Our 1st solution was to parallelise the string matching completely such that each index of the sample got its own thread, so the algorithm runs in O(length of signature). The code can be

found in the latest commit of the 1-pair-1-kernel branch. Then, for each (sample, string) pair, we launch a kernel to find the earliest match index. We used `cudaStreams` so that each kernel can run in parallel. However, it turned out to be quite slow, as launching kernels takes time and we have to launch about 1 million kernels. This is because launching kernels requires data transfer from device to gpu constant memory.

Test case 1: `./gen_sig 1000 10000 10000 0.001 > t1.fasta`
`./gen_sample t1.fasta 900 100 1 5 100000 100000 0 60 0.0001 > t1.fastq`

Node: a100-40



Appendix
Changing sequence length

Test command: `./gen_sig 1000 N N 0.001 > t1.fasta`
`./gen_sample t1.fasta 900 100 1 5 10*N 10*N 0 60 0. 001 > t1.fastq`

N: length of signature

Node: h100-96

Length of signature	Length of sample	Time
500	5000	0.4262
1000	10000	0.4605
2000	20000	0.4842
5000	50000	0.4611
10000	100000	0.6677

Changing percentage of signatures matching samples

Test command: `./gen_sig 1000 1000 1000 0.001 > t1.fasta`
`./gen_sample t1.fasta X Y 1 5 10000 10000 0 60 0.001 > t1.fastq`

X + Y = 1000, and they are varied to get the desired percentage of samples with virus signatures

Percentage	Time /s
10	0.3967
25	0.3810
50	0.3747
75	0.3817
90	0.3546

Changing fraction of N characters in sample

Node: a100-40
Test command: `./gen_sig 1000 3000 3100 0.1 > t1.fasta`
`./gen_sample t1.fasta 2000 20 1 1 10000 10000 10 10 X > t1.fastq`

Fraction	Time /s
0.1	3.752
0.2	4.230
0.3	4.729
0.4	5.559

Changing number of viruses
Node: a100-40
Test command: `./gen_sig 1000 3000 3100 0.1 > t1.fasta`
`./gen_sample t1.fasta 2000 20 X X 10000 10000 10 10 0.1 > t1.fastq`

Number	Time /s
1	3.591
5	3.635
10	3.685
15	3.638

Changing number of signatures

Node: a100-40
Test command: `./gen_sig X 3000 3100 0.1 > t1.fasta`
`./gen_sample t1.fasta 2000 20 1 1 10000 10000 10 10 0.1 > t1.fastq`

Signatures	Time /s
500	2.0750
600	2.2548
700	2.9485

800	2.9411
900	3.3527
1000	3.6173

Optimisation 1 raw data:

The code for unified memory is commit 130953b of branch 1-pair-1-thread, and for no unified memory at all, it is the latest commit of the same branch.

Node: h100-96

Testcase 1 command: ./gen_sig 1000 1000 1000 0.001 > t1.fasta
./gen_sample t1.fasta 900 100 1 5 10000 10000 0 60 0.0001 > t1.fastq

Using unified memory Average of 3 tries, time taken/s	No unified memory at all Average of 3 tries, time taken/s
0.3449	0.3500

Test case 2 command: ./gen_sig 1000 1000 1000 0.001 > t1.fasta
./gen_sample t1.fasta 900 100 1 5 100000 100000 0 60 0.0001 > t1.fastq

Using unified memory Average of 3 tries, time taken/s	No unified memory at all Average of 3 tries, time taken/s
1.003	0.9712

Test case 3 command: ./gen_sig 1000 10000 10000 0.001 > t1.fasta
./gen_sample t1.fasta 900 100 1 5 100000 100000 0 60 0.0001 > t1.fastq

Using unified memory Average of 3 tries, time taken/s	No unified memory at all Average of 3 tries, time taken/s
0.9235	0.9375

Optimisation 2 raw data:
Code for the 1st solution tried is in the latest commit of the 1-pair-1-kernel branch. Take average of 3 runs

Test case 1: ./gen_sig 1000 10000 10000 0.001 > t1.fasta
./gen_sample t1.fasta 900 100 1 5 100000 100000 0 60 0.0001 > t1.fastq

Node: a100-40

bench-a100	1st solution tried	Submitted solution
2.719	3.068	1.488